A study on the performance of reproducible computations

A study on the performance of reproducible computations

Nico Bombace[1] and Michèle Weiland[1]

EPCC, The University of Edinburgh, Bayes Centre, Edinburgh
{n.bombace,m.weiland}@epcc.ed.ac.uk

Abstract. Parallel computations are intrinsically non-reproducible, due to a combined effect of non-deterministic parallel reductions and non-associative floating point operations. Different strategies have been proposed in literature to alleviate this issue or eliminate it altogether, however at present there is no study on the performance impact of associative floating point operations on large scale applications. In this work, we implement associative operations using binned doubles in MiniFE, and perform various performance tests on Cirrus and Fulhame, two state-of-the-art HPC systems.

Keywords: Reproducibility · Binned doubles · Performance.

1 Introduction

High performance parallel computers play an increasingly impactful role in everyday life, aiding and accelerating new discoveries in science by leveraging parallel programming idioms. However, parallel computing poses a strong challenge regarding the reproducibility of results, due to the non-deterministic behaviour of parallel operations together with the non-associativity of floating point operations.

One of the direct consequences of this phenomenon is the challenge of verification of parallel applications or libraries, where code is often tested against an "oracle" solution to detect and eliminate software bugs as early as possible. However, if the run-to-run behaviour of the code cannot be guaranteed in terms of reproducibility, bugs can potentially be hidden even in production code [6].

Different methodologies have been proposed to tackle this problem, where the easiest approach is to freeze the summands before the operation, generally via sorting. While effective, this method is computationally expensive and efforts have been devoted to designing less expensive methodologies. For instance, compensated sum techniques [4] introduce an error accumulator which alleviates the non-reproducibility issue. A bit-reproducible computation of the results can be guaranteed by designing new floating point operations that are associative by definition.

In particular, this work will focus on the technique proposed by Ahrens et al. in [2], in which the notion of *binned doubles* is introduced. Preliminary studies

have shown that with respect to a standard double implementation, binned doubles introduce a slowdown of $4x$, however this is based only on the dot product operation. In our work, the proposed binned double is implemented in the Mantevo project [3], which is comprised of several mini-apps that represent different aspects of demanding computations. This study focuses on the MiniFE mini-app, an implicit finite element solver that supports the use of custom numerical types. With this set of operations we are able to perform an extensive study on the performance of reproducible operations. We show that:

1. Binned doubles ensure reproducibility of computations on different architectures,
2. It is possible to use the same code for both doubles and reproducible doubles using a non-invasive compile-time approach,
3. reproducibility requirements strongly affect performance.

2 Methodology

We provide a binned doubles implementation to use in the MiniFE application, which forms part of the Mantevo project [3], aimed at the analysis of HPC system performance. In particular, this mini-app provides a fully fledged implementation of unstructured finite elements, with support for multi-core operations.

2.1 MiniFE

MiniFE is written in C++ and relies heavily on the use of meta-programming techniques in the form of templates, which represent a compile-time mechanism to write generic code not coupled to a particular type [7]. As an example, let us consider the Vector declaration in MiniFE:

```
template<typename Scalar,
         typename LocalOrdinal,
         typename GlobalOrdinal>
struct
Vector{
  ...
  typedef Scalar        ScalarType;
  typedef LocalOrdinal  LocalOrdinalType;
  typedef GlobalOrdinal GlobalOrdinalType;
  ...
};
```

Listing 1.1. Definition of MiniFE vector.

In Listing 1.1 the type of scalar and the variables used to store the local and global indexes defined respectively by the template parameters `Scalar`, `LocalOrdinal` and `GlobalOrdinal` can be swapped at compile time. Moreover, the types of the template parameters can be retrieved at compile time using the variables `ScalarType`, `LocalOrdinalType` and `GlobalOrdinalType`. This

technique provides high design flexibility while avoiding performance issues of runtime techniques such as inheritance.

MiniFE supports scalar types which implement the standard $+, -, *, /$ operations as well as additional information required at compile-time: namely the `magnitude_type` required to compute the norm of the underlying numerical value, its `name` for report purposes and the `mpi_type()` used in MPI computations (not active in this work) . Such additional information can be provided through the specialization of ad-hoc template structures called traits. For example, the traits of a built-in C++ double type are defined as:

```
template<>
struct TypeTraits<double> {
  typedef double magnitude_type;
  static const char* name() {return "double";}
#ifdef HAVE_MPI
  static MPI_Datatype mpi_type() {return MPI_DOUBLE;}
#endif
};
```

Listing 1.2. Double type traits specialization.

In Listing 1.3, it can be noted that the type traits also contain the corresponding MPI datatype. As an example we show the use of such type traits to resolve, at compile time, the name of the type used in a vector:

```
char* dataName = TypeTraits<
                 Vector<Scalar,
                        LocalOrdinal,
                        GlobalOrdinal>::ScalarType>::name();
```

Listing 1.3. Use of type traits specialization.

MiniFE provides type traits for all C++ built in types, while user defined types can specialise the `TypeTraits` structure, as well as overload the basic $+, -, */$ operations.

2.2 Implementation of binned doubles

The original C implementation of binned doubles provided in [2] is modified and encapsulated in a C++ class template, `reproducible::Double<K>`, which calls the original ReproBLAS [2] functions using overloaded operators. The `K` parameter is an integer that defines the number of bins available for a double. Each `reproducible::Double<K>` contains two private member variables: the original double, and a `std::array<double, 2*K>` that contains its binned representation. One of the tests used to verify the implementation checks that the sum of all the elements in a vector of `reproducible:Double<K>` always yields the same result, no matter how the vector is shuffled. A code snippet of such a test is shown in Listing 1.4.

```
std::vector<reproducible::Double<3> > reproX;
std::random_device rd;
```

```
std::mt19937 g(rd());
...
auto reproSum1 = std::accumulate(reproX.begin(), reproX.end(), reproducible::
    Double<3>(0));
auto doubleReproSum1 = reproSum1.getDouble();
std::shuffle(std::begin(reproX), std::end(reproX), g);
auto reproSum2 = std::accumulate(reproX.begin(), reproX.end(), reproducible::
    Double<3>(0));
auto doubleReproSum2 = reproSum2.getDouble();
EXPECT_EQ(doubleReproSum1, doubleReproSum2);
```

Listing 1.4. Reproducible double Sum.

The `reproducible::Double<K>` and the associated `TypeTraits` cannot be readily used in the current OpenMP-parallel version of MiniFE, because it uses atomic and reduction clauses that do not support operator overloading [5]. To overcome this limitation we propose a modification of the MiniFE mini-app as follows:

- Substitution of atomic clauses with critical clauses;
- Substitution of reduction clauses, with a combination of a parallel private reduction and a critical accumulation section as shown in Listing 1.5.

```
#pragma omp parallel for
    reduction(+:result)
 for(int i=0; i<n; ++i) {
   result += xcoefs[i] * ycoefs[i];
}
```

```
#pragma omp parallel
{
  MINIFE_SCALAR result_private = 0;
  #pragma omp for nowait
  for(int i=0; i<n; ++i) {
    result_private += xcoefs[i] * ycoefs[i];
  }
  #pragma omp critical
  {
    result += result_private;
  }
}
```

Listing 1.5. Original and modified MiniFE comparison. On the left side, parallel reductions are accomplished using a reduction clause. On the right side, the same result is achieved using a "parallel for", which accumulates in a private variable, and subsequently a critical section accumulates into a global variable.

In order to assess that such changes did not alter the performance of the code, we benchmarked the original and the modified version of MiniFE using standard doubles and 180 8-nodes 3D hexahedral elements in every direction. We used one node of the Cirrus UK National Tier-2 HPC Service at EPCC [1], which has a total of 280 compute nodes, each with 256 GB of memory and two 2.1 GHz, 18-core Intel Xeon (Broadwell) processors, connected using an Infiband FDR network using a hyper-cube topology. The compiler used is GCC 6.3.0. The results are reported in Figure 1 and Figure 1 for Cirrus and Fulhame respectively.

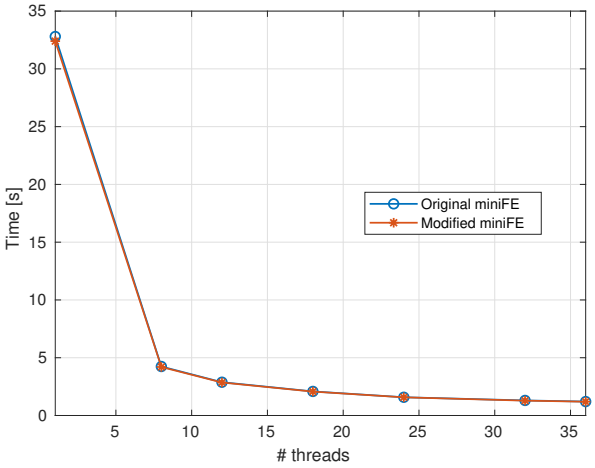

Fig. 1. Comparison of original and proposed modified version of MiniFE. The proposed modification do not alter the performance of the original code.

Figure 1 reports the runtime of the Conjugate Gradient solver in linear scale. It is clear that the modifications to the MiniFE mini-app do not change its performance. In the next section, the run times are compared against the same application using binned doubles.

3 Comparison of doubles against binned doubles

The modifications to MiniFE provided in the previous section allow the use different scalar types used by MiniFE at compile time, using the `TypeTraits` structure and the class template `reproducible::Double<K>`. We therefore compiled and ran MiniFE using `reproducible::Double<3>` and the inputs described in the previous section. Note that for the reproducible MiniFE, all the doubles in the application are substituted for binned doubles. To verify cross-platform reproducibility we compiled and ran MiniFe on one node of the Fulhame system at EPCC, a 64-node Arm-based HPE Apollo70 system, with two 32-core Marvell ThunderX2 processors and 256GB memory per node. It uses an Infiniband EDR interconnect with a non-blocking fat tree topology.

3.1 Performance comparison

The performance comparisons are reported in Figures 2, 3, Figures **??**, and Tables 1, 2 for the Cirrus and Fulhame system respectively. There is an average slowdown of $36x$ in terms of runtime when using binned doubles, instead of built-in doubles on Intel architecture, while this gap is reduced to $26x$ on our Arm-based system. The log-log plot (Figures 2 shows that the two versions (non-reproducible and reproducible) of MiniFE follow similar performance trends.

The same information is reported in tabular form in Tables 1 and 2. In terms of speedup (Figure 3 and Figure 4), the reproducible MiniFE exhibits slight better scaling when increasing the number of cores.

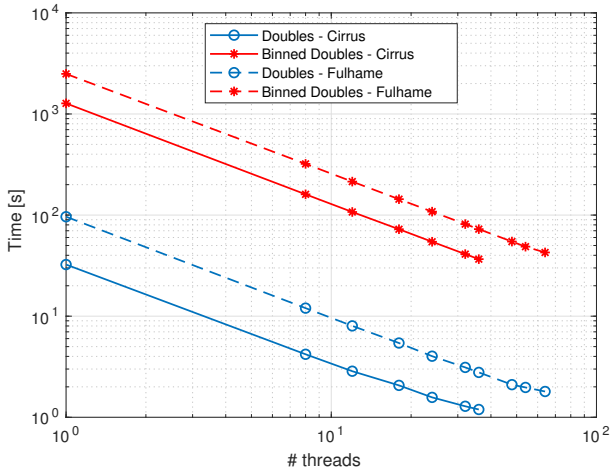

Fig. 2. Comparison of running time of Conjugate Gradient solver in log-log scale for built-in doubles and binned doubles with $K = 3$.

	non-reproducible MiniFE			reproducible MiniFE		
Threads	Time [s]	Residual Norm	N Iterations	Time [s]	Residual Norm	N Iterations
1	63.3068	0.2115	391	4193.2	0.1984	350
8	7.5857	0.2089	365	533.7	0.1984	350
12	5.1138	0.2060	363	356.4	0.1984	350
18	3.6689	0.2028	361	238.3	0.1984	350
24	2.8148	0.1970	357	179.4	0.1984	350
32	2.2967	0.1963	359	134.9	0.1984	350
36	2.1214	0.1960	357	120.3	0.1984	350

Table 1. Comparison between non-reproducible and reproducible Mini-FE on Cirrus using GCC and -O0 compiler flag. The residual is scaled by 1^{-15}.

3.2 Reproducibility

Figure 5 compares the final number of iterations necessary to achieve a prescribed residual norm obtained with doubles (non-reproducible MiniFE) and binned-doubles (reproducible MiniFE), using the aforementioned version of GCC compiler, with -O0 compiler flag on Cirrus and Fulhame. The error-bars indicate

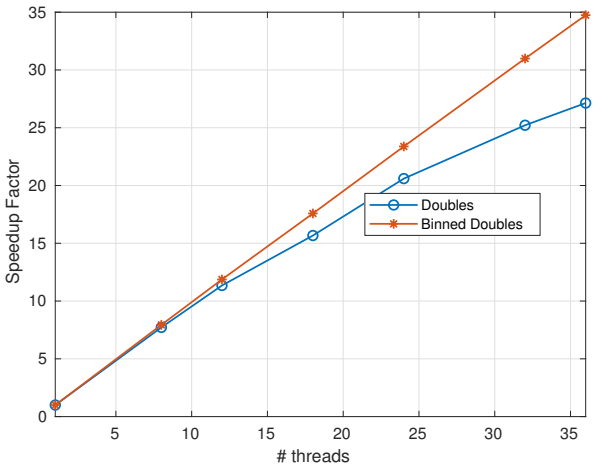

Fig. 3. Comparison of speedup factor of Conjugate Gradient solver for built-in doubles and binned doubles with $K = 3$ on Cirrus.

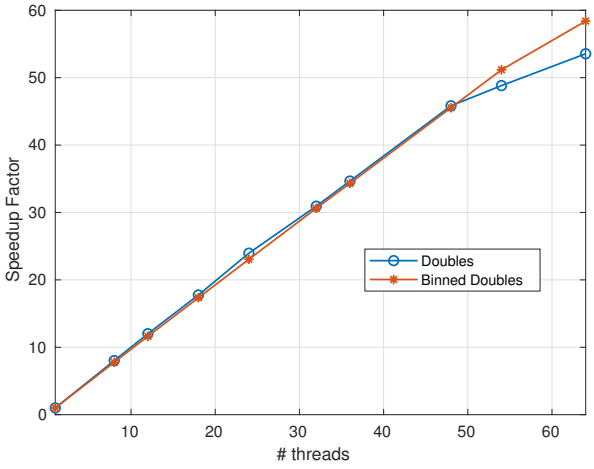

Fig. 4. Comparison of speedup factor of Conjugate Gradient solver for built-in doubles and binned doubles with $K = 3$ on Fulhame.

	non-reproducible MiniFE			reproducible MiniFE		
Threads	Time [s]	Residual Norm	N Iterations	Time [s]	Residual Norm	N Iterations
1	186.6910	0.2115	391	4193.2	0.1984	350
8	22.7928	0.2067	365	533.7	0.1984	350
12	15.3523	0.2047	364	356.4	0.1984	350
18	10.4003	0.2165	361	238.3	0.1984	350
24	7.7301	0.1966	356	179.4	0.1984	350
32	5.9495	0.2015	358	134.9	0.1984	350
36	5.2481	0.1960	358	120.3	0.1984	350
48	3.9851	0.1962	356	90.7	0.1984	350
54	3.7412	0.1968	356	80.8	0.1984	350
64	3.3409	0.1960	356	71.7	0.1984	350

Table 2. Comparison between non-reproducible and reproducible Mini-FE on Fulhame using GCC and -O0 compiler flag. The residual is scaled by 1^{-15}.

a 95% confidence interval under the hypothesis of normal distribution, obtained processing five runs at each fixed number of processes. When using binned doubles, the final number of iterations of the finite element problem is constant and does not depend on the number of threads or architecture model. When using built-in doubles however, such values are not constant, and importantly their behaviour is also not predictable. The same information is reported in tabular form in Tables 1 and 2. An important detail is given by the serial run of MiniFE using the standard doubles. In this case the final number of iterations is indeed reproducible due to the absence of the non deterministic effect of parallel reductions .

Figure 6 compares the final number of iterations on the same application but built supplying -O3 flag to the compiler. Differently from the previous case, there is an effect of the optimization on different platforms, even in the serial run for standard doubles. However, when using binned doubles, the results confirm that these yield reproducible results, independent of the degree of parallelism, platform architecture and build process.

Finally, we investigated the reproducibility of binned doubles when using different compilers. In particular, we built MiniFE, using both GCC and Clang compilers on Fulhame. The results shown in Figure 7 demonstrate that standard doubles reproducibility, contrarily to binned doubles, is affected by the use of different compilers,

4 Discussion of results

The performance results listed earlier show that using binned doubles results in is a significant performance hit. This is not unexpected of course as binned doubles are not a native type and crucial compiler optimisations will not be applied to them in the same way as built-in doubles. However, an important feature of the approach taken here is its flexibility. The choice of which numerical type to

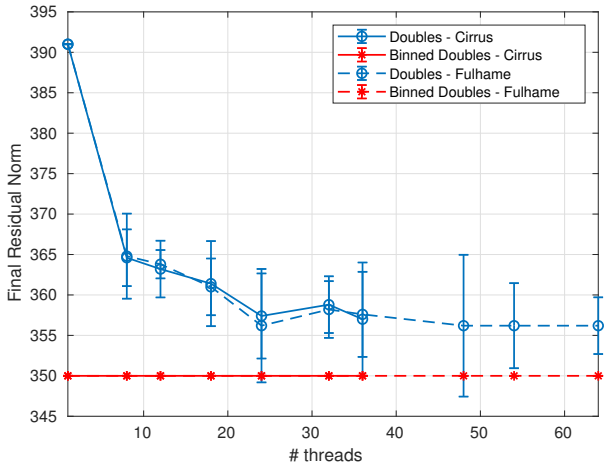

Fig. 5. Comparison of final number of Iterations for built-in doubles and binned doubles with $K = 3$, on Cirrus and Fulhame with GCC and -O0 flag.

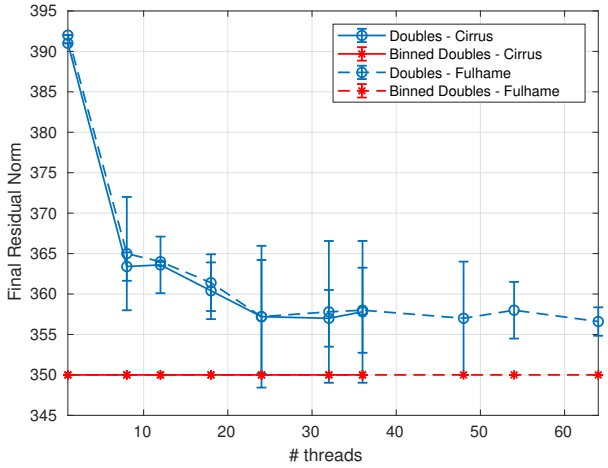

Fig. 6. Comparison of final number of Iterations for built-in doubles and binned doubles with $K = 3$, on Cirrus and Fulhame with GCC and -O3 flag.

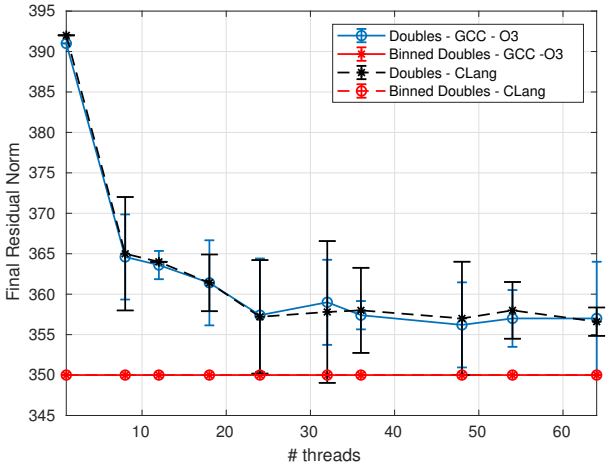

Fig. 7. Comparison of final number of Iterations for built-in doubles and binned doubles with $K = 3$, on Fulhame with GCC and Clang compilers.

use is taken only at compile time, and it therefore does not impact the software development process. In this context therefore, binned doubles might be used in the development of new features and functionalities for existing software. In particular, we would propose the use of binned doubles as part of the feature testing and verification process of parallel software development, where performance is a secondary requirement with respect to correctness. Once a code's correctness has been verified, it can be used in production with the default built-in doubles compile flag enabled. The negative impact on performance is acceptable in that scenario.

As mentioned earlier, it is worth pointing out that the current version of MiniFe uses only one numerical type for all the computation phases, which means that using the compiler flags system the compiler will "blindly" substitute and swap the numerical types. This strategy is very aggressive, and we suggest the use of a multiple compile flags system, where binned doubles are used only in critical parts of the code. This will decrease the performance penalty while still achieving reproducible results.

Finally, although a slowdown on the order on 30-40x is not acceptable for production runs, the use of binned doubles is nevertheless not prohibitively expensive for full application verification tests. In an environment where testing the correctness of a full application is critically important, using binned doubles remains an option.

5 Conclusions and further work

In this paper we have shown the use of binned doubles in a finite element application using the MiniFE proxy application. The proposed approach uses a

compile type parameter to determine the use of built-in doubles or binned doubles. We have validated a modification to the original MiniFE application to make it compatible with types that overload the arithmetic operators $+, -, *, /$. Subsequently, we have created a new C++ class template in the reproducible namespace `reproducible::Double<K>`, which wraps the functionalities of the ReproBLAS library. The major advantage of this methodology is that the user of MiniFE can easily switch between customised and built-in types, through the change of a compile flag, effectively choosing between reproducible and non-reproducible computations.

While we have experienced a noticeable slowdown in terms of performance, the use of binned doubles has indeed achieved results which are reproducible (as evidenced by the final residuals) and independent of the number of parallel threads or architecture. This result establishes the use of binned doubles in the verification process of the development cycle, where reproducible results are a more stringent requirement than performance. Since the proposed methodology does not require alteration of the source code, but only affects the compilation flags, in other phases of the development cycle the numerical type used can be reverted to built-in doubles, which will deliver results in a fraction of the time (but non-reproducible).

To lower the impact on performance of the binned doubles, we are currently investigating a mixed double/binned doubles approach, in which binned doubles will only be used in parts of the code where reproducibility is lost, due to non-deterministic behaviour of reduction operations or vectorisation. Moreover, we are investigating the use and impact of reproducible doubles in distributed and heterogeneous parallel environments, using MPI and mixed-mode parallelism.

6 Acknowledgement

This work used the Cirrus UK National Tier-2 HPC Service at EPCC, funded by the University of Edinburgh and EPSRC (EP/P020267/1). The Fulhame system is supplied to the EPCC as part of the Catalyst UK program, a collaboration with Hewlett Packard Enterprise, Arm and SUSE to accelerate the adoption of Arm based supercomputer applications in the UK.

References

1. Cirrus UK National Tier-2 HPC service (2019), http://www.cirrus.ac.uk
2. Ahrens, P., Nguyen, H.D., Demmel, J.: Efficient reproducible floating point summation and blas. Tech. Rep. UCB/EECS-2015-229, EECS Department, University of California, Berkeley (Dec 2015)
3. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving Performance via Mini-applications. Tech. Rep. SAND2009-5574, Sandia National Laboratories (2009)
4. Kahan, W.: Pracniques: further remarks on reducing truncation errors. Communications of the ACM **8**(1), 40 (1965)

5. OpenMP Architecture Review Board: OpenMP application program interface version 3.1 (2001), https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf
6. Robey, R.W., Robey, J.M., Aulwes, R.: In search of numerical consistency in parallel programming. Parallel Computing **37**(4-5), 217–229 (2011)
7. Vandevoorde, D., Josuttis, N.M.: C++ Templates. Addison-Wesley Longman Publishing Co., Inc. (2002)